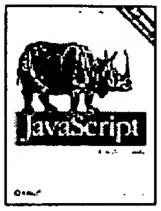


JavaScript: The Definitive Guide, 3rd Edition

By David Flanagan

Publisher: **O'Reilly**
Pub Date: **June 1998**
More recent edition of this book available.
ISBN: **1-56592-392-8**
Pages: **790**
Slots: **1.0**

09/577,190

Table of Contents | Index | Examples | Errata | Additional Reading

Chapter 1. Introduction to JavaScript

JavaScript is a lightweight interpreted programming language with object-oriented capabilities. The general-purpose core of the language has been embedded in Netscape Navigator, Internet Explorer, and other web browsers and embellished for web programming with the addition of objects that represent the web browser window and its contents. This client-side version of JavaScript allows executable content to be included in web pages—it means that a web page need no longer be static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

Syntactically, the core JavaScript language resembles C, C++, and Java, with programming constructs such as the `if` statement, the `while` loop, and the `&&` operator. The similarity ends with this syntactic resemblance, however. JavaScript is an untyped language, which means that variables do not need to have a type specified. Objects in JavaScript are more like Perl's associative arrays than they are like structures in C or objects in C++ or Java. The object-oriented inheritance mechanism of JavaScript is like that of the little-known languages Self and NewtonScript, which is quite different from inheritance in C++ and Java. Like Perl, JavaScript is an interpreted language, and it draws inspiration from Perl in a number of places, such as its regular expression and array-handling features.

This chapter is a quick overview of JavaScript; it explains what JavaScript can and cannot do and exposes some myths about the language. It distinguishes the core JavaScript language from embedded and extended versions of the language, such as the client-side JavaScript that is embedded in web browsers and the server-side JavaScript that is embedded in web servers. (This book documents core and client-side JavaScript.) This chapter also demonstrates real-world web programming with some client-side JavaScript examples.

URL http://proquest.safaribooksonline.com/1565923928/jscript-ch-intro

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

Section Title	Book Title
1. Introduction to JavaScript	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2. Versions of JavaScript	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
	Webmaster in a Nutshell, 3rd Edition

3. JavaScript	By Robert Eckstein, Stephen Spainhour
4. The JavaScript Language	JavaScript Pocket Reference, 2nd Edition By David Flanagan
5. JavaScript in Web Browsers	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
6. JavaScript	Webmaster in a Nutshell, 2nd Edition By Robert Eckstein, Stephen Spainhour
7. JavaScript in Web Browsers	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
8. Client-side JavaScript	JavaScript Pocket Reference, 2nd Edition By David Flanagan
9. Using the Rest of This Book	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
10. What JavaScript Is	JavaScript™ by Example By Ellie Quigley

1.1. JavaScript Myths

JavaScript is the subject of a fair bit of misinformation and confusion. Before proceeding any further with our exploration of JavaScript, it is important that we debunk some common and persistent myths about the language.

1.1.1. JavaScript Is Not Java

One of the most common misconceptions about JavaScript is that it is a simplified version of Java, the programming language from Sun Microsystems. Other than an incomplete syntactic resemblance and the fact that both Java and JavaScript can provide executable content in web browsers, the two languages are entirely unrelated. The similarity of names is purely a marketing ploy (the language was originally called LiveScript and its name was changed to JavaScript at the last minute).

JavaScript and Java do, however, make a good team. The two languages have disjoint sets of capabilities. JavaScript can control browser behavior and content but cannot draw graphics or perform networking. Java has no control over the browser as a whole but can do graphics, networking, and multithreading. Client-side JavaScript can interact with and control Java applets embedded in a web page, and in this sense, JavaScript really can script Java.[1]

[1] Navigator 3 and later versions support a technology known as LiveConnect, which goes beyond merely interacting with applets and allows much more comprehensive scripting of Java. Chapter 20, describes LiveConnect in detail.

1.1.2. JavaScript Is Not Simple

JavaScript is touted as a scripting language instead of a programming language, the implication being that scripting languages are simpler, that they are programming languages for non-programmers. Indeed, JavaScript appears at first glance to be a fairly simple language, perhaps of the same complexity as BASIC. The language does have a number of features designed to make it more forgiving and easier to use for new and unsophisticated programmers. Non-programmers can use JavaScript for limited, cookbook-style programming tasks.

Beneath its thin veneer of simplicity, however, JavaScript is a full-featured programming language, as complex as any, and more complex than some. Programmers who attempt to use JavaScript for non-trivial tasks often find the process frustrating if they do not have a solid understanding of the language. Therefore, this book documents JavaScript comprehensively so that you can develop a sophisticated understanding of the language.

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

	Section Title	Book Title
1.	JavaScript Myths	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2.	Getting Started with JavaScript	Special Edition Using JavaScript By Paul McFedries
3.	The JavaScript Language	Platinum Edition Using XHTML™, XML, and Java™ 2 By Eric Ladd, Jim O'Donnell, Mike Morgan, Andrew H. Watt
4.	Introduction to JavaScripting	Platinum Edition Using XHTML™, XML, and Java™ 2 By Eric Ladd, Jim O'Donnell, Mike Morgan, Andrew H. Watt
5.	Getting into JavaScript!	JavaScript™ 1.5 by Example By Adrian Kingsley-Hughes, Kathie Kingsley-Hughes
6.	Using the Rest of This Book	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
7.	Learning Web Scripting Basics	Sams Teach Yourself JavaScript™ in 24 Hours, Second Edition By Michael Moncur
8.	Introduction to JavaScripting	Platinum Edition Using HTML 4, XML, and Java 1.2
9.	Handling Different Versions of JavaScript	Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions By Budi Kurniawan
10.	Exploring JavaScript's Capabilities	Sams Teach Yourself JavaScript™ in 24 Hours, Second Edition By Michael Moncur

1.2. Versions of JavaScript

T he JavaScript language is still evolving, and several different versions of JavaScript are available. The original version of the language, now almost obsolete, is JavaScript 1.0. The next version, JavaScript 1.1, is much more robust and introduces some important new features, such as improved support for arrays. As of this writing, the current version of the language is JavaScript 1.2, which contains a number of new features, like support for regular expressions, the switch statement, and the delete operator. The upcoming version of the language is expected to be called JavaScript 1.3, and it is likely to have still more new features, such as support for exception handling (an advanced error handling and recovery feature found in languages like C++ and Java).

The name JavaScript is owned by Netscape. Microsoft's implementation of the language is officially known as JScript. Versions of JScript are more or less compatible with the equivalent versions of JavaScript, although JScript skipped a version and went directly from JavaScript 1.0 compatibility to JavaScript 1.2 compatibility.

JavaScript has been standardized by the European Computer Manufacturers Association (ECMA) and is on a fast track for standardization by the International Organization for Standardization (ISO). The relevant standards are ECMA-262, and, when standardized by ISO, ISO-16262. These standards define a language officially known as ECMAScript, which is approximately equivalent to JavaScript 1.1, although not all implementations of JavaScript currently conform to all details of the ECMA standard. The name ECMAScript is universally regarded as ugly and cumbersome and was chosen precisely for this reason: it favors neither Netscape's JavaScript nor Microsoft's JScript. In this book, I use the term ECMA-262 or simply ECMA to refer to the standardized version of the language. As of this writing, proposals for a second, updated version of the standard have been submitted to ECMA by Netscape and Microsoft. The process is still in its early stages, however, and it will be some time before ECMA-262 Version 2 becomes standardized.

In addition to the various versions of the core JavaScript language, variants exist for each context that JavaScript is embedded in. The following sections describe some of these contexts.

1.2.1. Client-Side JavaScript

When a JavaScript interpreter is embedded in a web browser, the result is client-side JavaScript. This is by far the most common variant of JavaScript; when most people refer to JavaScript, they usually mean client-side JavaScript. This book documents client-side JavaScript, along with the core JavaScript language that client-side JavaScript incorporates.

We'll discuss client-side JavaScript and its capabilities in much more detail later in this chapter. In brief, though, client-side JavaScript combines the scripting ability of a JavaScript interpreter with the document object model defined by a web browser. These two distinct technologies combine in a synergistic way, so the result is greater than the sum of its parts: client-side JavaScript enables executable content to be distributed over the web and is at the heart of a new generation of "Dynamic HTML" documents.

As with the core language itself, there are a number of different versions of client-side JavaScript. Different browser versions incorporate different versions of the JavaScript interpreter. Table 1.1 shows the core language version supported by different versions of the two major web browsers.

Table 1.1. Versions of JavaScript Supported by Various Browsers

	Browser Name	
Browser Version	**Netscape Navigator**	**Microsoft Internet Explorer**
2	JavaScript 1.0	
3	JavaScript 1.1	JavaScript 1.0
4	JavaScript 1.2; not fully ECMA-262 compliant	JavaScript 1.2; ECMA-262 compliant

The differences and incompatibilities between Netscape's and Microsoft's client-side versions of JavaScript are much greater than the differences between their respective implementations of the core language. Still, there is a large subset of client-side JavaScript features that both browsers agree upon. For lack of better names, versions of client-side JavaScript are typically referred to by the version of the core language they are based on. Thus, in many contexts, the term JavaScript 1.2 refers to the version of client-side JavaScript supported by Navigator 4 and Internet Explorer 4. When I use core-language version numbers to refer to client-side versions of JavaScript, I am referring to the compatible subset of features supported by Navigator and Internet Explorer. When I discuss client-side features specific to one browser or the other, I will refer to the browser by name and version number.

Note that Navigator and Internet Explorer are not the only browsers that support client-side JavaScript. For example, Opera 3 (www.operasoftware.com), supports client-side JavaScript as well. However, since Navigator and Internet Explorer have the vast majority of market share, they are the only browsers discussed explicitly in this book. Client-side JavaScript implementations in other browsers should conform fairly closely to the implementations in these two browsers.

Similarly, JavaScript is not the only programming language that can be embedded within a web browser. For example, Internet Explorer also supports a language known as VBScript. This variant of Microsoft's Visual Basic language provides many of the same features as JavaScript but can only be used with Microsoft browsers.[2] The HTML 4.0 specification uses the Tcl programming language as an example of an embedded scripting language in its discussion of the HTML <SCRIPT> tag. While there are no mainstream browsers that support Tcl for this purpose, there is no reason that a browser could not easily support this language.

[2] However, NCompass makes a Navigator plugin called ScriptActive that allows VBScript to work in Navigator (www.ncompasslabs.com).

Readers will notice that this book covers Navigator more thoroughly than Internet Explorer. This bias towards Netscape has steadily declined in each subsequent edition of the book, as Microsoft's implementation of client-side JavaScript has matured and as Internet Explorer has gained market share, but the bias still exists in this edition. The primary reason for the bias is that Netscape invented JavaScript. Until standardization efforts are complete and the language stops evolving so rapidly, Netscape's implementation must naturally be regarded as more definitive and leading-edge than Microsoft's. On the other hand, as we'll see in Chapter 14, and elsewhere, there are important areas in which Microsoft's implementation of client-side JavaScript is much closer to emerging standards than Netscape's is.

1.2.2. Server-Side JavaScript

Later in this chapter, you'll see how the core JavaScript language has been extended for use in web browsers. Netscape has also taken the core language and extended it in an entirely different way for use in web servers. Netscape initially called their server-side JavaScript product "LiveWire" (not to be confused with LiveConnect, documented in Chapter 20, or with LiveScript, which was the original name for JavaScript), but now they simply refer to it as server-side JavaScript. Microsoft also supports server-side programming with JavaScript in its web server, using its Active Server Pages (ASP) framework. Since this is a simple overview of server-side scripting, we only discuss Netscape's im plementation of server-side JavaScript here.

Server-side JavaScript provides an alternative to CGI scripts. It goes beyond the CGI model, in fact, because server-side JavaScript code is embedded directly within HTML pages, which allows executable server-side scripts to be directly intermixed with web content. Whenever a document containing server-side JavaScript code is requested by the client, the server executes the script or scripts contained in the document and sends the resulting document (which may be partially static and partially dynamically generated) to the requester. Because execution speed is a very important issue on production web servers, HTML files that contain server-side JavaScript are precompiled to a binary form that can be more efficiently interpreted and sent to the requesting client.

An obvious capability of server-side JavaScript is to dynamically generate HTML to be displayed by the client. Its most powerful features, however, come from the server-side objects it has access to. The File object, for example, allows a server-side script to read and write files on the server. And the Database object allows scripts to perform SQL database queries and updates.

Besides the File and Database objects, server-side JavaScript also provides other powerful objects, including the Request and Client objects. The Request object encapsulates information about the current HTTP request that the server is processing. This object contains any query string or form values that were submitted with the request, for example. The Client object has a longer lifetime than the Request object and allows a server-side JavaScript script to save state across multiple HTTP requests from the same client. Because this object provides such an easy way to save state between requests, writing programs with server-side JavaScript feels much different from writing simple CGI scripts. In fact, the Client object makes it feasible to go beyond writing scripts and create web applications easily.

Because server-side JavaScript is, at least at this point, a proprietary vendor-specific server-side technology, and because it is used by fewer programmers than client-side JavaScript, it is not documented in this book. Nevertheless, the chapters that discuss the core JavaScript language will still be valuable to server-side JavaScript programmers.

1.2.3. Embedded JavaScript

JavaScript is a general-purpose programming language; its use is not restricted to web browsers and web servers. Although web browsers and servers are currently the only highly visible applications that add scripting capabilities by embedding a JavaScript interpreter, there are certainly many other applications in which such a scripting capability would be very useful. Both Netscape and Microsoft have made their JavaScript interpreters available to companies and to programmers that want to embed them in their applications. In addition, Netscape has made the source code of its JavaScript interpreter freely available, along with all the source code to its web browser, on the www.mozilla.org web site.

Also, there is a project underway at Netscape to produce a JavaScript implementation written in Java.[3] Such an implementation would allow virtually any Java program to include JavaScript scripting capabilities quite easily.

[3] Flatteringly enough, the codename for this Netscape project is "Rhino."

Finally, Netscape and Microsoft are not the only players in the embedded JavaScript market. For example, a company named Nombas (www.nombas.com) produces and sells its own commercial JavaScript interpreter, designed specifically to be embedded within applications. Also, Ribbit Software Systems (www.ribbit-soft.com) produces a commercial JavaScript interpreter and compiler, both written in Java.

We can expect to see more and more applications that use JavaScript as an embedded scripting language. If you are writing scripts for such an application, you'll find the first half of this book, documenting the core language, to be useful. The web-browser specific chapters, however, will probably not be applicable to your scripts.

URL http://proquest.safaribooksonline.com/1565923928/jscript3-ID-1.2

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow..

	Section Title	Book Title
1.	JavaScript	Webmaster in a Nutshell, 3rd Edition By Robert Eckstein, Stephen Spainhour
2.	JavaScript	Webmaster in a Nutshell, 2nd Edition By Robert Eckstein, Stephen Spainhour
3.	Introduction to JavaScript	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
4.	JavaScript in Web Browsers	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
5.	Client-Side JavaScript	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
6.	JavaScript in Web Browsers	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
7.	What JavaScript Is	JavaScript™ by Example By Ellie Quigley
8.	understanding javascript	The Unusually Useful Web Book By June Cohen
9.	Client-side JavaScript	JavaScript Pocket Reference, 2nd Edition By David Flanagan
10.	Introduction to JavaScript	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan

1.4. Client-Side JavaScript Features

JavaScript is a general-purpose programming language, and, as such, you can write programs in it to perform arbitrary computations. You can write simple scripts, for example, that compute Fibonacci numbers, or search for primes. In the context of the Web and web browsers, however, a more interesting application of the language would be a program that computed the sales tax on an online order, based on information supplied by the user in an HTML form. As mentioned earlier, the real power of JavaScript lies in the browser and document-based objects that the language supports. To give you an idea of JavaScript's potential, the following sections list and explain the important capabilities of client-side JavaScript and the objects it supports.

1.4.1. Control Document Appearance and Content

The JavaScript Document object, through its write() method, which we have already seen, allows you to write arbitrary HTML into a document as the document is being parsed by the browser. For example, this allows you to include the current date and time in a document or to display different content on different platforms.

You can also use the Document object to generate documents entirely from scratch. Properties of the Document object allow you to specify colors for the document background, the text, and for the hypertext links within it. What this amounts to is the ability to generate dynamic and conditional HTML documents, a technique that works particularly well in multiframe documents. Indeed, in some cases, dynamic generation of frame content allows a JavaScript program to replace a traditional CGI script entirely.

The new technology of Dynamic HTML is based on the ability to use JavaScript to dynamically modify the contents and appearance of HTML documents. Internet Explorer 4 supports a complete document object model that gives JavaScript access to every single HTML element within a document. In addition, Internet Explorer's DHTML capabilities allow JavaScript to modify the content of any element and to change the appearance of the element dynamically by modifying its style sheet properties.[5] As you can imagine, this nearly complete power over an HTML document has tremendous dynamic potential. Client-side JavaScript is at the core of this potential.

[5] Navigator 4 does not support as complete a DOM, but a future version of Navigator will support the DOM standard now emerging from the W3C standardization process.

1.4.2. Control the Browser

Several JavaScript objects allow control over the behavior of the browser. The Window object supports methods to pop up dialog boxes to display simple messages to the user and to get simple input from the user. This object also defines a method to create and open (and close) entirely new browser windows, which can have any specified size and any combination of user controls. This allows you, for example, to open up multiple windows to give the user multiple views of your web site. New browser windows are also useful for temporary display of generated HTML, and, when created without the menubar and other user controls, these windows can serve as dialog boxes for more complex messages or user input.

JavaScript does not define methods that allow you to create and manipulate frames directly within a browser window. However, the ability to generate HTML dynamically allows you to write programmatically the HTML tags that create any desired frame layout.

JavaScript also allows control over which web pages are displayed in the browser. The Location object allows you to download and display the contents of any URL in any window or frame of the browser. The History object allows you to move forward and back within the user's browsing history, simulating the action of the browser's **Forward** and **Back** buttons.

Yet another method of the Window object allows JavaScript to display arbitrary messages to the user in the status line of any browser window.

1.4.3. Interact with HTML Forms

Another important aspect of client-side JavaScript is its ability to interact with HTML forms. This capability is provided by the Form object and the Form element objects it can contain: Button, Checkbox, Hidden, Password, Radio, Reset, Select, Submit, Text, and Textarea objects. These element objects allow you to read and write the values of the input elements in the forms in a document. For example, an online catalog might use an HTML form to allow the user to enter his order and could use JavaScript to read his input from that form in order to compute the cost of the order, the sales tax, and the shipping charge. JavaScript programs like this are, in fact, very common on the Web. We'll see a program shortly that uses an HTML form and JavaScript to allow the user to compute monthly payments on a home mortgage or other loan. JavaScript has an obvious advantage over server-based CGI scripts for applications like these: JavaScript code is executed on the client, so the form's contents don't have to be sent to the server in order for relatively simple computations to be performed.

Another common use of client-side JavaScript with forms is for verification of a form before it is submitted. If client-side JavaScript is able to perform all necessary error checking of a user's input, the CGI script on the server side can be much simpler and, more importantly, there is no round trip to the server to detect and inform the user of the errors. Client-side JavaScript can also perform preprocessing of input data, which can reduce the amount of data that must be transmitted to the server. In some cases, client-side JavaScript can eliminate the need for CGI scripts on the server altogether! (On the other hand, JavaScript and CGI do work well together. For example, a CGI program can dynamically create JavaScript code on the fly, just as it dynamically creates HTML.)

1.4.4. Interact with the User

An important feature of JavaScript is the ability to define event handlers—arbitrary pieces of code to be executed when a particular event occurs. Usually, these events are initiated by the user, when, for example, she moves the mouse over a hypertext link, enters a value in a form, or clicks the **Submit** button in a form. This event-handling capability is a crucial one, because programming with graphical interfaces, such as HTML forms, inherently requires an event-driven model. JavaScript can trigger any kind of action in response to user events. Typical examples might be to display a special message in the status line when the user positions the mouse over a hypertext link or to pop up a confirmation dialog box when the user submits an important form.

1.4.5. Read and Write Client State with Cookies

"Cookie" is Netscape's term for a small amount of state data stored permanently or temporarily by the client. Cookies are transmitted to and from the server and allow a web page or web site to "remember" things about the client—for example, that the user has previously visited the site, or has already registered and obtained a password, or has expressed a preference about the color and layout of web pages. Cookies help you provide the state information that is missing from the stateless HTTP protocol of the Web.

When cookies were invented, they were intended for use exclusively by CGI scripts; although stored on the client, they could only be read or written by the server. The idea was to allow a CGI script to generate and send different HTML to the client depending on the value of a cookie (or cookies). JavaScript changes this because JavaScript programs can read and write cookie values, and as I noted earlier in this chapter, they can dynamically generate HTML based on the value of cookies. The implications of this are subtle. CGI programming is still an important technique in many cases that use cookies. In some cases, however, JavaScript can entirely replace the need for CGI.

1.4.6. Still More Features

In addition to the features I have already mentioned, JavaScript has many other capabilities:

- As of JavaScript 1.1, you can change the image displayed by an <IMG> tag. This allows sophisticated effects, such as having an image change when the mouse passes over it or when the user clicks on a button elsewhere in the browser.
- As of JavaScript 1.1, JavaScript can interact with Java applets and other embedded objects that appear in the browser. JavaScript code can read and write the properties of these applets and objects and can also invoke any methods they define. This feature truly allows JavaScript to script Java.
- As mentioned at the start of this section, JavaScript can perform arbitrary computation. JavaScript has a floating-point data type, arithmetic operators that work with it, and a full complement of standard floating-point mathematical functions.
- The JavaScript Date object simplifies the process of computing and working with dates and times.
- The Document object supports a property that specifies the last modified date for the current document. You can use it to display a timestamp on any document automatically.
- JavaScript has a `window.setTimeout()` method that allows a block of arbitrary JavaScript code to be executed some number of milliseconds in the future. This is useful for building delays or repetitive actions into a JavaScript program. In JavaScript 1.2, `setTimeout()` is augmented by another useful method called `setInterval()`.
- The Navigator object (named after the web browser, of course) has variables that specify the name and version of the browser that is running, as well as variables that identify the platform it is running on. These variables allow scripts to customize their behavior based on browser or platform, so that they can take advantage of extra capabilities supported by some versions or work around bugs that exist on some platforms.
- In client-side JavaScript 1.2, the Screen object provides information about the size and color-depth of the monitor on which the web browser is being displayed.
- As of JavaScript 1.1, the `scroll()` method of the Window object allows JavaScript programs to scroll windows in the X and Y dimensions. In JavaScript 1.2, this method is augmented by a host of others that allow browser windows to be moved and resized.

1.4.7. What JavaScript Can't Do

Client-side JavaScript has an impressive list of capabilities. Note, however, that they are confined to browser-related and HTML-related tasks. Since client-side JavaScript is used in a limited context, it does not have features that would be required for standalone languages:

- JavaScript does not have any graphics capabilities, except for the powerful ability to generate HTML dynamically (including images, tables, frames, forms, fonts, etc.) for the browser to display.
- For security reasons, client-side JavaScript does not allow the reading or writing of files. Obviously, you

wouldn't want to allow an untrusted program from any random web site to run on your computer and rearrange your files!

- JavaScript does not support networking of any kind, except that it can cause the browser to download arbitrary URLs and it can send the contents of HTML forms to CGI scripts, email addresses, and Usenet newsgroups.

URL http://proquest.safaribooksonline.com/1565923928/jscript3-ID-1.4

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

	Section Title	Book Title
1.	Client-Side JavaScript Features	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2.	Client-Side JavaScript: Executable Content in Web Pages	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
3.	Client-Side JavaScript: Executable Content in Web Pages	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
4.	Using the Rest of This Book	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
5.	Client-side JavaScript	JavaScript Pocket Reference, 2nd Edition By David Flanagan
6.	Checking Whether JavaScript Is Enabled	Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions By Budi Kurniawan
7.	Versions of JavaScript	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
8.	JavaScript in Web Browsers	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
9.	Using the Rest of This Book	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
10.	JavaScript	Platinum Edition Using XHTML™, XML, and Java™ 2 By Eric Ladd, Jim O'Donnell, Mike Morgan, Andrew H. Watt

Chapter 12. JavaScript in Web Browsers

T he first part of this book described the core JavaScript language. Now we move on to JavaScript as used within web browsers, commonly called client-side JavaScript.[1] Most of the examples we've seen until now, while legal JavaScript code, had no particular context—they were JavaScript fragments that ran in no specified environment. This chapter provides that context. It begins with a conceptual introduction to the web browser programming environment and to basic client-side JavaScript concepts. Next, it discusses how we actually embed JavaScript code within HTML documents so that it can run in a web browser. Finally, the chapter goes into detail about how JavaScript programs are executed in a web browser.

[1] The term "client-side JavaScript" is left over from the days when JavaScript was used in only two places: web browsers (clients) and web servers. As JavaScript is adopted as a scripting language in more and more places, the term "client-side" will no longer make much sense because it doesn't specify the client-side of *what*. Nevertheless, we'll continue to use the term in this book.

URL http://proquest.safaribooksonline.com/1565923928/jscript-ch-client

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

Section Title	Book Title
1. JavaScript in Web Browsers	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2. Versions of JavaScript	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
3. Client-side JavaScript	JavaScript Pocket Reference, 2nd Edition By David Flanagan
4. Introduction to JavaScript	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
5. JavaScript	Webmaster in a Nutshell, 3rd Edition By Robert Eckstein, Stephen Spainhour

6. Introduction to JavaScript	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
7. What Can't You Do with JavaScript?	Special Edition Using JavaScript By Paul McFedries
8. Client-Side JavaScript	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
9. understanding javascript	The Unusually Useful Web Book By June Cohen
10. What JavaScript Is	JavaScript™ by Example By Ellie Quigley

12.1. The Web Browser Environment

To understand client-side JavaScript, you must understand the conceptual framework of the programming environment provided by a web browser. The following sections introduce three important features of that programming environment:

- The Window object that serves as the global object and global execution context for client-side JavaScript code
- The client-side object hierarchy
- The event-driven programming model

12.1.1. The Window as Global Execution Context

The primary task of a web browser is to display HTML documents in a window. In client-side JavaScript, the Document object represents an HTML document, and the Window object represents the window (or frame) that displays the document. While the Document and the Window objects are both important to client-side JavaScript, the Window object is more important. This is for one substantial reason: the Window object is the global object in client-side programming.

Recall from Chapter 4, that in every implementation of JavaScript there is always a global object at the head of the scope chain; the properties of this global object are global variables. In client-side JavaScript, the Window object is the global object. The Window object defines a number of properties and methods that allow us to manipulate the web browser window. It also defines properties that refer to other important objects, such as the document property for the Document object. Finally, the Window object has two self-referential properties, window and self. You can use either of these global variables to refer directly to the Window object.

Since the Window object is the global object in client-side JavaScript, all global variables are defined as properties of the window. For example, the following two lines of code perform essentially the same function:

```
var answer = 42;    // Declare and initialize a global variable.
window.answer = 42; // Create a new property of the Window object.
```

The Window object represents a web browser window or a frame within a window. To client-side JavaScript, top-level windows and frames are essentially equivalent. It is common to write JavaScript applications that use multiple frames and possible, if less common, to write applications that use multiple windows. Each window or frame involved in an application has a unique Window object and defines a unique execution context for client-side JavaScript code. In other words, a global variable declared by JavaScript code in one frame is not a global variable within a second frame. However, the second frame *can* access a global variable of the first frame; we'll see how when we consider these issues in more detail in Chapter 13.

12.1.2. The Client-Side Object Hierarchy

We've seen that the Window object is the key object in client-side JavaScript. All other client-side objects are connected to this object. For example, every Window object contains a `document` property that refers to the Document object associated with the window and a `location` property that refers to the Location object associated with the window. A window object also contains a `frames[]` array that refers to the Window objects that represent the frames of the original window. Thus, `document` represents the Document object of the current window, and `frames[1].document` refers to the Document object of the second child frame of the current window.

An object referenced through the current window or through some other Window object may itself refer to other objects. For example, every Document object has a `forms[]` array containing Form objects that represent any HTML forms appearing in the document. To refer to one of these forms, you might write:

```
self.document.forms[0]
```

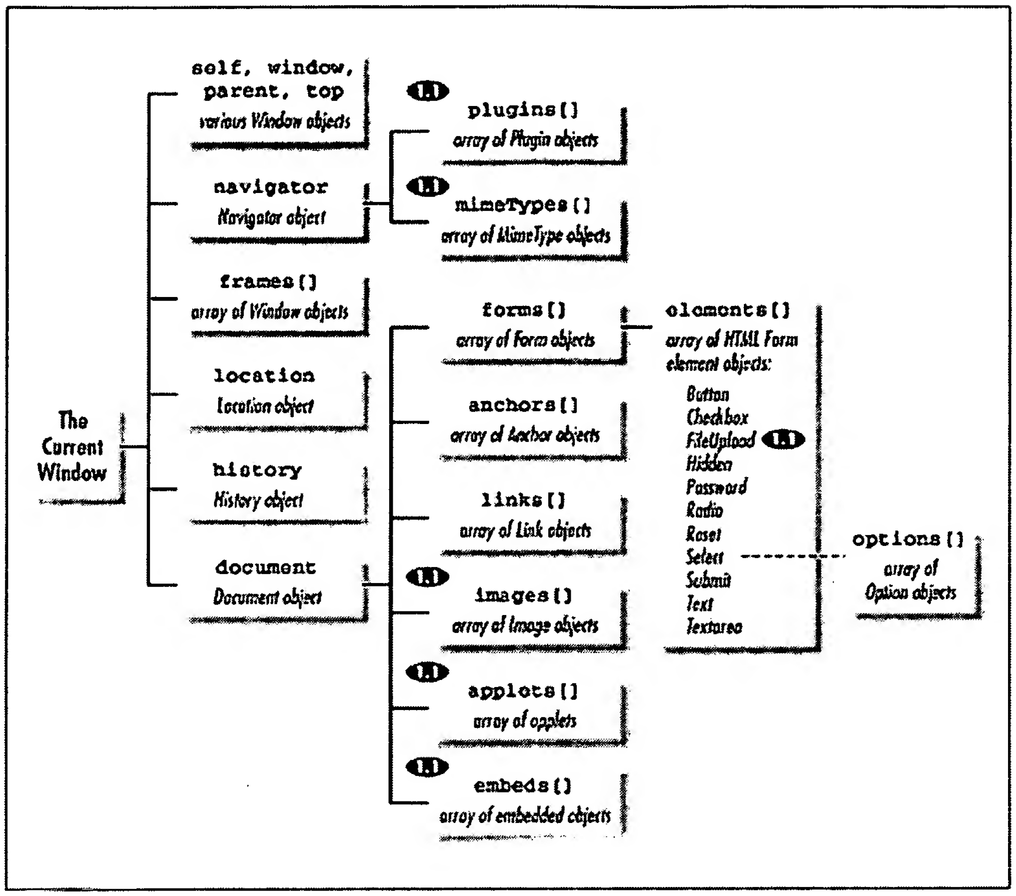
To continue with the same example, each Form object has an `elements[]` array containing objects that represent the various HTML form elements (input fields, buttons, etc.) that appear within the form. In extreme cases, you can write code that refers to an object at the end of a whole chain of objects, ending up with expressions as complex as this one:

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

We've seen that the Window object is the global object at the head of the scope chain and that all client-side objects in JavaScript are accessible as properties of other objects. What this means is that there is a hierarchy of JavaScript objects with the Window object at its root. Figure 12.1 shows this hierarchy. Study this figure carefully; understanding the hierarchy and the objects it contains is crucial to successful client-side JavaScript programming.

Note that Figure 12.1 shows just the object properties that refer to other objects. Most of the objects shown in the diagram have quite a few more properties than those shown. The notation "1.1" in the figure indicates properties that were added in JavaScript 1.1. Later chapters document the objects shown in the object hierarchy diagram and demonstrate common JavaScript programming techniques that employ those objects. You may find it useful to refer back to this figure while reading these chapters.

Figure 12.1. The JavaScript object hierarchy



12.1.3. The Event-Driven Programming Model

In the old days, computer programs often ran in batch mode. This meant that they read in a batch of data, did some computation on that data, and then wrote out the results. Later, with timesharing and text-based terminals, limited kinds of interactivity became possible—the program could ask the user for input, and the user could type in data. The computer could then process the data and display the results on screen.

Nowadays, with graphical displays and pointing devices like mice, the situation is different. Programs are generally event driven; they respond to asynchronous user input in the form of mouse clicks and keystrokes in a way that depends on the position of the mouse pointer. A web browser is just such a graphical environment. An HTML document contains an embedded GUI (graphical user interface), so client-side JavaScript uses the event-driven programming model.

It is perfectly possible to write a static JavaScript program that does not accept user input and does exactly the same thing every time. Sometimes this sort of program is useful. More often, however, we want to write dynamic programs that interact with the user. To do this, we must be able to respond to user input.

In client-side JavaScript, the web browser notifies our programs of user input by generating *events*. There are various types of events, such as keystroke events, mouse motion events, and so on. When an event occurs, the web browser attempts to invoke an appropriate *event handler* function to respond to the event. Thus, to write dynamic, interactive client-side JavaScript programs, we must define appropriate event handlers and register them with the system, so that the browser can invoke them at appropriate times.

If you are not already accustomed to the event-driven programming model, it can take a little getting used to. In the old model, you would write a single, monolithic block of code that followed some well-defined flow of control and ran to completion from beginning to end. Event-driven programming stands this model on its head. In event-driven p rogramming, you write a number of independent (but mutually interacting) event handlers. You do not invoke these handlers directly, but allow the system to invoke them at the appropriate times. Since they are triggered by the user's input, the handlers will be invoked at unpredictable, asynchronous times. Much of the time your program is not running at all, but merely sitting waiting for the system to invoke one of its event handlers.

The next section explains how JavaScript code is embedded within HTML files. It shows how we can define both static blocks of code that run synchronously from start to finish and also event handlers that are invoked asynchronously by the system. We'll also discuss events and event handling in much greater detail in Chapter 15.

URL http://proquest.safaribooksonline.com/1565923928/jscript3-ID-12.1

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

	Section Title	Book Title
1.	The Web Browser Environment	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2.	Windows and Frames	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
3.	Revisiting the Window Object Hierarchy	Special Edition Using JavaScript By Paul McFedries
4.	The Window Object	JavaScript Pocket Reference, 2nd Edition By David Flanagan
5.	Window	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
6.	The window Object	Platinum Edition Using XHTML™, XML, and Java™ 2 By Eric Ladd, Jim O'Donnell, Mike Morgan, Andrew H. Watt
7.	Client-Side JavaScript Features	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
8.	Window	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
9.	Working With Browser Windows	JavaScript for the World Wide Web, 4th Edition: Visual QuickStart Guide By Tom Negrino, Dori Smith
10.	Using Window Events	Platinum Edition Using XHTML™, XML, and Java™ 2 By Eric Ladd, Jim O'Donnell, Mike Morgan, Andrew H. Watt

12.2. Embedding JavaScript in HTML

I f JavaScript is to be integrated into a web browser, an obvious requirement is that JavaScript code be embedded into the documents that web browsers display. There are actually seven ways that JavaScript code can be embedded into HTML documents:

- Between a pair of <SCRIPT> and </SCRIPT> tags
- From an external file specified by the SRC or ARCHIVE attributes of a <SCRIPT> tag
- In an event handler, specified as the value of an HTML attribute such as onClick or onMouseOver
- As the body of a URL that uses the special javascript: protocol
- In a style sheet, between <STYLE TYPE="text/javascript" > and </STYLE> tags
- In a JavaScript entity, as the value of an HTML attribute
- In a conditional comment that comments out HTML text unless a given JavaScript expression evaluates to true

The following sections document each of these seven JavaScript embedding techniques in more detail. Together, they explain all the ways to include JavaScript in web pages—that is, they explain the allowed structure of JavaScript programs on the client side.

12.2.1. The <SCRIPT> Tag

Client-side JavaScript scripts are part of an HTML file and are usually coded within the <SCRIPT> and </SCRIPT> tags. You may place any number of JavaScript statements between these tags; they are executed in order of appearance, as part of the document loading process. <SCRIPT> tags may appear in either the <HEAD> or <BODY> of an HTML document.

A single HTML document may contain any number of non-overlapping pairs of <SCRIPT> and </SCRIPT> tags. These multiple, separate scripts are executed in the order in which they appear within the document. While separate scripts within a single file are executed at different times during the loading and parsing of the HTML file, they constitute part of the same JavaScript program: functions and variables defined in one script are available to all scripts that follow in the same file.[2] For example, you can have the following script somewhere in an HTML page:

[2] In Navigator 4, the exception to this rule is that scripts located in their own layers (within <LAYER> tags, for example) run in their own contexts.

```
<SCRIPT>var x = 1;</SCRIPT>
```

Later on in the same HTML page, you can refer to x, even though it's in a different script block. The context that matters is the HTML page, not the script block:

```
<SCRIPT>document.write(x);</SCRIPT>
```

Example 12.1 shows a sample HTML file that includes a simple JavaScript program. Note the difference between this example and many of the code fragments shown earlier in the book: this one is integrated with an HTML file and has a clear context in which it runs. Note also the use of a LANGUAGE attribute in the <SCRIPT> tag. This is explained in the next section.

Example 12.1. A Simple JavaScript Program in an HTML File

```
<HTML>
<HEAD>
<TITLE>Today's Date</TITLE>
  <SCRIPT LANGUAGE="JavaScript">
  // Define a function for use later on.
  function print_todays_date()
  {
    var d = new Date();  // Today's date and time
    document.write(d.toLocaleString());
  }
  </SCRIPT>
</HEAD>
<BODY>
<HR>The date and time are:<BR><B>
  <SCRIPT LANGUAGE="JavaScript">
  // Now call the function we defined above.
  print_todays_date();
  </SCRIPT>
</B><HR>
</BODY>
</HTML>
```

12.2.1.1. The LANGUAGE attribute

The <SCRIPT> tag has an optional LANGUAGE attribute that specifies the scripting language used for the script. This attribute is necessary because there is more than one version of JavaScript and there is more than one scripting language that can be embedded between <SCRIPT> and </SCRIPT> tags. By specifying which language a script is written in, you tell a browser whether it should attempt to interpret the script, or whether the script is written in a language that the browser doesn't understand, and should therefore be ignored.

If you are writing JavaScript code, use the LANGUAGE attribute as follows:

```
<SCRIPT LANGUAGE="JavaScript">
  // JavaScript code goes here.
</SCRIPT>
```

On the other hand, if you are writing a script in Microsoft's VBScript language[3] you should use the attribute like this:

[3] The language is actually called Visual Basic Scripting Edition. Obviously, it is a version of Microsoft's Visual Basic language. The only browser that supports VBScript is Internet Explorer. VBScript interfaces with HTML objects in the same way that JavaScript does, but the core language itself has a different syntax than JavaScript.

```
<SCRIPT LANGUAGE="VBScript">
  ' VBScript code goes here (' is a comment character like // in JavaScript)
</SCRIPT>
```

When you specify the LANGUAGE="JavaScript" attribute for a script, any JavaScript-enabled browser will run the script, while browsers that understand the <SCRIPT> tag but do not understand JavaScript will ignore the script. But the LANGUAGE attribute can be used to specify more than just the scripting language in use: it can also specify the version of the language. For example, if you specify LANGUAGE="JavaScript1.1", only browsers that support JavaScript 1.1 (or later) will run the script; other browsers (such as Navigator 2) will ignore it. Similarly, if you specify a LANGUAGE attribute of "JavaScript1.2", only JavaScript 1.2 browsers (such as Navigator 4) will run the script.

The use of the string "JavaScript1.2" in the LANGUAGE attribute deserves special mention. When Navigator 4 was being prepared for release, it appeared that the emerging ECMA-262 standard would require some incompatible changes to certain features of the language. To prevent these incompatible changes from breaking existing scripts, the designers of JavaScript at Netscape took the sensible precaution of implementing the changes only when "JavaScript1.2" was explicitly specified in the LANGUAGE attribute. This ensured that only code written explicitly for the new platform would get the new behavior. JavaScript code written for previous browsers would still get the old-style behavior it expected.

Unfortunately, the ECMA standard was not finalized before Navigator 4 was released, and after the release, the proposed incompatible changes to the language were removed from the standard. Thus, specifying a LANGUAGE attribute of "JavaScript1.2" makes Navigator 4 behave in ways that are not compatible with previous browsers and not compatible with the ECMA specification. For this reason, you may want to avoid specifying "JavaScript1.2" as a value for the LANGUAGE attribute.

JavaScript is, and is likely to remain, the *default* scripting language for the Web. If you omit the LANGUAGE attribute, both Navigator and Internet Explorer default to the value "JavaScript". Nonetheless, because there are now multiple scripting languages available, it is a good habit to always use the LANGUAGE attribute to specify exactly what language (or what version) your scripts are written in.

12.2.1.2. The </SCRIPT> tag

You may at some point find yourself writing a script that writes a script into some other browser window or frame.[4] If you do this, you'll need to write out a </SCRIPT> tag to terminate the script you are writing. You must be careful, though—the HTML parser doesn't know about quoted strings, so if you write out a string that contains the characters "</SCRIPT>", the HTML parser terminates the currently running script.

[4] This happens more often than you might think: one commonly used feature of JavaScript is the ability to dynamically generate HTML and JavaScript content for display in other browser windows and frames.

To avoid this problem, simply break the tag up into pieces and write it out using an expression like "</"+"SCRIPT>":

```
<SCRIPT>
f1.document.write("<SCRIPT>");
f1.document.write("document.write('<H2>This is the quoted script</H2>')");
f1.document.write("</" + "SCRIPT>");
</SCRIPT>
```

Alternatively, you can escape the / in </SCRIPT> with a backslash:

```
f1.document.write("<VSCRIPT>");
```

12.2.2. Including JavaScript Files

As of JavaScript 1.1, the <SCRIPT> tag supports a SRC attribute. The value of this attribute specifies the URL of a file of JavaScript code. It is used like this:

```
<SCRIPT SRC="../../javascript/util.js"></SCRIPT>
```

A JavaScript file is just that—pure JavaScript, without <SCRIPT> tags or any other HTML. A JavaScript file typically has a *.js* extension, and should be exported by a web server with MIME-type application/x-javascript. This last point is important; your web server may require special configuration in order to successfully use JavaScript files in this way.

The <SCRIPT> tag with the SRC attribute specified behaves exactly as if the contents of the specified JavaScript file appeared directly between the <SCRIPT> and </SCRIPT> tags. Any code that does appear between the open and close <SCRIPT> tags is ignored by browsers that support the SRC attribute (although it is still executed by browsers like Navigator 2 that do not recognize the tag). Note that the closing </SCRIPT> tag is required even when the SRC attribute is specified and there is no JavaScript between the <SCRIPT> and </SCRIPT> tags.

There are a number of advantages to using the SRC tag:

- It simplifies your HTML files by allowing you to remove large blocks of JavaScript code from them.
- When you have a function or other JavaScript code used by several different HTML files, you can keep it in a single file and read it into each HTML file that needs it. This reduces disk usage and makes code maintenance much easier.
- When JavaScript functions are used by more than one page, placing them in a separate JavaScript file allows them to be cached by the browser, making them load more quickly. When JavaScript code is shared by multiple pages, the time savings of caching more than outweigh the small delay required for the browser to open a separate network connection to download the JavaScript file the first time it is requested.
- Because the SRC attribute takes an arbitrary URL as its value, a JavaScript program or web page from one web server can employ code (such as subroutine libraries) exported by other web servers.

12.2.2.1. The ARCHIVE attribute

In Navigator 4, the SRC attribute of the <SCRIPT> tag is complemented by the ARCHIVE attribute. ARCHIVE specifies a JAR (Java archive) file that contains a number of compressed JavaScript files (and may also contain other auxiliary files, such as digital signatures). If your program uses a number of JavaScript files, it can be more efficient to combine them into a single compressed JAR file that can be loaded over the network. Note that the ARCHIVE attribute specifies only the name of the archive, not the name of the individual *.js* file that you want to use within it. Thus, the ARCHIVE attribute must be used with the SRC attribute. For example:

```
<SCRIPT ARCHIVE="utils.jar" SRC="animation.js"></SCRIPT>
```

A JAR archive file is simply a common ZIP file with some additional manifest information added. Netscape provides a free tool that allows developers to create JAR archives. One of the most important uses of archives is attaching digital signatures to scripts. We'll talk more about this, and about creating and using JAR archives, in Chapter 21.

12.2.3. Event Handlers

JavaScript code in a <SCRIPT> is executed once, when the HTML file that contains it is read into the web browser. A program that uses only this sort of static script cannot respond dynamically to the user. More dynamic programs define event handlers that are automatically invoked by the web browser when certain events occur—for example, when the user clicks on a button within a form. Because events in client-side JavaScript originate from HTML objects (like buttons), event handlers are defined as attributes of those objects.

In order to allow us to define JavaScript event handlers as part of HTML object definitions, JavaScript extends HTML[5] by adding new event handler attributes to various HTML tags. For example, to define an event handler that is invoked when the user clicks on a checkbox in a form, you specify the handler code as an attribute of the HTML tag that defines the checkbox:

[5] These event handler extensions to HTML have now been recognized and standardized by the HTML 4.0 standard.

```
<INPUT
  TYPE="checkbox"

  NAME="opts"
  VALUE="ignore-case"
  onClick="ignore_case = this.checked;"
>
```

What's of interest to us here is the onClick attribute.[6] The string value of the onClick attribute may contain one or more JavaScript statements. If there is more than one statement, the statements must be separated from each other with semicolons. When the specified event—in this case, a click—occurs on the checkbox, the JavaScript code within the string is executed.

[6] All event handler attribute names begin with "on." The mixed-case capitalization of onClick is a common convention for JavaScript event handlers defined in HTML files. HTML element and attribute names are case-insensitive, but writing "onClick" rather than "ONCLICK" sets off the handlers from standard HTML tags, which are, by convention, shown in all capitals.

While you can include any number of JavaScript statements within an event handler definition, a common technique when more than one or two simple statements are required is to define the body of an event handler as a function between <SCRIPT> and </SCRIPT> tags. Then you can simply invoke this function from the event handler. This keeps most of your actual JavaScript code within scripts and reduces the need to mingle JavaScript and HTML.

We'll cover events and event handlers in much more detail in Chapter 15.

12.2.3.1. Event handlers in <SCRIPT> tags

In Internet Explorer, but not in Navigator, there is an alternative syntax for defining event handlers. It involves using new FOR and EVENT attributes to the <SCRIPT> tag to specify code that constitutes an event handler for a named object and a named event. Using this Internet Explorer technique, we could rewrite the checkbox example shown earlier like this:

```
<INPUT TYPE="checkbox" NAME="opts" VALUE="ignore-case">
<SCRIPT FOR="opts" EVENT="onClick">
  ignore_case = this.checked;
</SCRIPT>
```

Note that the value of the FOR attribute must be an object name assigned with the NAME attribute when the object is defined. And the value of the EVENT attribute is the name of the event handler (but not the name of the event itself).

There is a certain elegance to specifying event handlers in this way--it avoids the need to add new JavaScript-specific attributes to all HTML objects. However, this technique for defining event handlers is typically of more use to VBScript programmers than to JavaScript programmers, and since it is not supported by Navigator, its use is not recommended.

12.2.4. JavaScript in URLs

Another way that JavaScript code can be included on the client side is in a URL following the javascript: pseudo-protocol specifier. This special protocol type specifies that the body of the URL is arbitrary JavaScript code to be run by the JavaScript interpreter. If the JavaScript code in a javascript: URL contains multiple statements, the statements must be separated from one another by semicolons. Such a URL might look like this:

```
javascript:var now = new Date(); "<h1>The time is:</h1>" + now;
```

When the browser loads one of these JavaScript URLs, it executes the JavaScript code contained in the URL and displays the document referred to by the URL. This document is the string value of the last JavaScript statement in the URL. The string is formatted and displayed just like any other document loaded into the browser.

More commonly, a JavaScript URL contains JavaScript statements that perform actions but return no value. For example:

```
javascript:alert("Hello World!")
```

When this sort of URL is loaded, the browser executes the JavaScript code, but because there is no value to display as the new document, it does not modify the currently displayed document.

Note that as of JavaScript 1.1, you can use the void operator to force an expression to have no value. This is useful when you want to execute an assignment statement, for example, but do not want to display the assigned value in the browser window. (Recall that assignment statements are also expressions, and that they evaluate to the value of the right-hand side of the assignment.)

A javascript: URL can be used anywhere you'd use a regular URL. In Navigator, one important way to use this syntax is to type it directly into the **Location** field of your browser, where it allows you to test arbitrary JavaScript code without having to get out your editor and create an HTML file containing the code.

In fact, Navigator takes this idea even further. As described in Chapter 1, if you enter the URL javascript: alone, with no JavaScript code following it, Navigator displays a JavaScript interpreter page that allows you to sequentially enter and execute lines of code. Unfortunately, neither of these techniques works in Internet Explorer.

javascript: URLs can also be used in other contexts. You might use one as the target of a hypertext link, for example. When the user clicks on the link, the specified JavaScript code is executed. Or, if you specify a javascript: URL as the value of the ACTION attribute of a <FORM> tag, the JavaScript code in the URL is executed when the user submits the form. In these contexts, the javascript: URL is essentially a substitute for an event handler. Event handlers and javascript: URLs can often be used interchangeably—which you choose is a stylistic matter.

There are a few circumstances where a javascript: URL can be used with objects that do not support event handlers. For example, the <AREA> tag does not support an onClick event handler on Windows platforms in Navigator 3 (though it does in Navigator 4). So, if you want to execute JavaScript code when the user clicks on a client-side image map prior to Navigator 4, you must use a javascript: URL.

12.2.5. JavaScript Style Sheet Syntax

In Navigator 4, JavaScript code may appear in a style sheet, between the tag <STYLE TYPE="text/javascript" > and the tag </STYLE>. Any such JavaScript code should appear in the <HEAD> of an HTML document; its purpose should be to define a style sheet by setting properties of the tags, classes, and ids attributes. JavaScript style sheet (JSS) syntax is an alternative to standard cascading style sheet (CSS) syntax. For example, the following style sheet uses JavaScript code to specify that all <H1> headings in a document should appear in bold red text:

```
<STYLE TYPE="text/javascript">
 tags.H1.fontstyle="bold";
 tags.H1.color = "red";
</STYLE>
```

Although JavaScript code in a STYLE tag should only be used to defined style sheets, in practice it can be used (or abused) for any purpose, like a <SCRIPT> tag. The only difference between <STYLE> code and <SCRIPT> code is that JavaScript code in a <STYLE> tag is scoped to the Document object rather than the Window object. That is, when it looks up variables, it finds properties (such as tags) of the Document object before it finds properties of the Window object. Because this is the only difference, it is also possible to define style sheet syntax with a <SCRIPT> tag. Note, however, that we must explicitly specify that the tags and other objects are properties of the Document object:

```
<SCRIPT LANGUAGE="javascript">
 document.tags.H1.fontstyle="bold";
 document.tags.H1.color = "red";
</SCRIPT>
```

Note that because JSS syntax is not supported by Internet Explorer and is not likely to be adopted as a standard, its use is discouraged. In general, you should use standard CSS syntax wherever possible. We'll see more about cascading style sheet and JavaScript style sheet syntax in Chapter 17.

12.2.6. JavaScript Entities

In Navigator 3 and later, JavaScript code may appear in HTML attribute values in a special form known as a *JavaScript entity*. Recall that an HTML entity is a sequence of characters like &lt; that represents a special character like <. A JavaScript entity is similar. It has the following syntax:

```
&{ JavaScript-statements };
```

The entity may contain any number of JavaScript statements, which must be separated from one another by semicolons. It must begin with an ampersand and an open curly bracket and end with a close curly bracket and a semicolon.

Whenever an entity is encountered in HTML, it is replaced with its value. The value of a JavaScript entity is the value of the last JavaScript statement or expression within the entity, converted to a string.

In general, entities can be used anywhere within HTML code. JavaScript entities, however, may appear only within the values of HTML attributes. These entities allow you, in effect, to write conditional HTML. Typical usage might look like this:

```
<BODY BGCOLOR="&{favorite_color();};">
<INPUT TYPE="text" NAME="lastname" VALUE="&{defaults.lastname};">
```

12.2.7. Conditional Comments

In Navigator 4 and later, a JavaScript entity may be used with a modified HTML comment syntax to produce a *conditional comment*. If the JavaScript expression embedded in the entity evaluates to true, the comment is ignored and the body of the comment (typically HTML text) is processed as usual. If the expression evaluates to false (or is not evaluated by a browser that doesn't understand this special comment syntax), the comment behaves as normal and the contents of the comment are ignored.

Conditional comments allow you to write JavaScript code that runs only on platforms that can support it. The code below, for example, runs only if the navigator.platform property (new in JavaScript 1.2) is equal to the string "win95" (i.e., if the code is running on a Windows 95 browser):

```
<!--&{navigator.platform == "win95"};
  <SCRIPT>
    ... // JavaScript code goes here.
  </SCRIPT>
-->
```

Unfortunately, since conditional comments are not supported by Internet Explorer 4, their utility is somewhat limited.

URL http://proquest.safaribooksonline.com/1565923928/jscript3-ID-12.2

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

Section Title	Book Title
1. Embedding JavaScript in HTML	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2. JavaScript in HTML	JavaScript Pocket Reference, 2nd Edition By David Flanagan
3. JavaScript in HTML	JavaScript Pocket Reference By David Flanagan
4. JavaScript in HTML	Webmaster in a Nutshell, 2nd Edition By Robert Eckstein, Stephen Spainhour
5. JavaScript	HTML & XHTML: The Definitive Guide, 5th Edition By Bill Kennedy, Chuck Musciano
6. JavaScript	HTML & XHTML: The Definitive Guide, 4th Edition By Bill Kennedy, Chuck Musciano
7. Introduction to JavaScript	Platinum Edition Using XHTML™, XML, and Java™ 2 By Eric Ladd, Jim O'Donnell, Mike Morgan, Andrew H. Watt
8. Basic Script Construction	Special Edition Using JavaScript By Paul McFedries
9. Scripting	Building Web Applications with UML
10. Scripting	Building Web Applications with UML Second Edition By Jim Conallen

12.3. Execution of JavaScript Programs

T he previous section discussed the mechanics of integrating JavaScript code into an HTML file. Now we move on to discuss exactly how that integrated JavaScript code is executed by the JavaScript interpreter. The following sections explain how different forms of JavaScript code are executed. While some of this material is fairly obvious, there are a number of important details that are not so obvious.

12.3.1. Scripts

JavaScript statements that appear between <SCRIPT> and </SCRIPT> tags are executed in order of appearance, and when more than one script appears in a file, those scripts are executed in the order in which they appear. The same rules apply to scripts included from separate files with the SRC attribute. This much is obvious.

The detail that is not so obvious but that is important to remember is that execution of scripts occurs as part of the web browser's HTML parsing process. Thus, if a script appears in the <HEAD> section of an HTML document, none of the <BODY> section of the document has been defined yet. This means that the JavaScript objects that represent the contents of the document body, such as Form and Link, have not been created yet and cannot be manipulated by that code.

Your scripts should not attempt to manipulate objects that haven't been created yet. For example, you can't write a script that manipulates the contents of an HTML form if the script appears before the form in the HTML file. Some other, similar rules apply on a case-by-case basis. For example, there are properties of the Document object that may be set only from a script in the <HEAD> section of an HTML document before Navigator has begun to parse the document content in the <BODY> section. Any special rules of this sort are documented in the reference entry for the affected object or property.

Since scripts are executed while the HTML document that contains them is being parsed and displayed, they should not take too long to run. An HTML document cannot be fully displayed until all scripts it contains have finished executing. If a script performs some computationally intensive task that takes a long time to run, the user may become frustrated waiting for the document to be displayed. Thus, if you need to perform a lot of computation with JavaScript, you should define a function to do the computation and invoke that function from an event handler when the user requests it rather than doing the computation when the document is first loaded.

As noted earlier, scripts that use the SRC attribute to read in external JavaScript files are executed just like scripts that include their code directly in the file. What this means is that the HTML parser and the JavaScript interpreter must both stop and wait for the external JavaScript file to be downloaded—scripts cannot be downloaded in parallel, as embedded images can. Downloading an external file of JavaScript code, even over a relatively fast modem connection, can cause noticeable delays in the loading and execution of a web page. Of course, once the JavaScript code is cached locally, this problem effectively disappears.

As we discussed in the previous section, Navigator 4 allows JavaScript code to be included in a style sheet within a pair of <STYLE> and </STYLE> tags. The JavaScript statements within a style sheet execute just like JavaScript statements within a <SCRIPT>, except that the scope chain is modified so that variables are looked up as properties of the Document object before they are looked up as properties of the Window object. This means that JavaScript style sheets can refer to the tags, classes, and ids properties of the Document object as if they were global variables, instead of using expressions like document.tags.

In Navigator 2, there is a notable bug relating to execution of scripts: whenever the web browser is resized, all the scripts within it are reinterpreted.

12.3.2. Functions

Remember that defining a function is not the same as executing it. It is perfectly safe to define a function that manipulates objects that haven't been created yet. You simply must take care that the function is not executed or invoked until the necessary variables, objects, and so on, all exist. I said earlier that you can't write a script to manipulate an HTML form if the script appears before the form in the HTML file. You can, however, write a script that defines a function to manipulate the form, regardless of the relative location of the script and form. In fact, this is a common thing to do. Many JavaScript programs start off with a script at the beginning of the file that does nothing more than define functions that are used further down in the HTML file.

It is also common to write JavaScript programs that use scripts simply to define functions that are later invoked through event handlers. As we'll see in the next section, you must take care in this case to ensure two things: that all functions are defined before any event handler attempts to invoke them, and that event handlers and the functions they invoke do not attempt to use objects that have not been defined yet.

12.3.3. Event Handlers

As we've seen, defining an event handler creates a JavaScript function. These event handler functions are defined as part of the HTML parsing process, but like functions defined directly by scripts, event handlers are not executed immediately. Event handler execution is asynchronous. Since events generally occur when the user interacts with HTML objects, there is no way to predict when an event handler will be invoked.

Event handlers share an important restriction with scripts: they should not take a long time to execute. As we've seen, scripts should run quickly because the HTML parser cannot continue parsing until the script finishes executing. Event handlers, on the other hand, should not take long to run because the user cannot interact with your program until the program has finished handling the event. If an event handler performs some time-consuming operation, it may appear to the user that the program has hung, frozen up, or crashed.

If for some reason you must perform a long operation in an event handler, be sure that the user has explicitly requested that operation and then be sure to notify him that there will be a wait. As we'll see in Chapter 13, you can notify the user by posting an alert() dialog box or by displaying text in the browser's status line. Also, if your program requires a lot of background processing, you can schedule a function to be called repeatedly during idle time with the setTimeout() method.

It is important to understand that event handlers may be invoked before a web page is fully loaded and parsed. This is easier to understand if you imagine a slow network connection—even a half-loaded document may display hypertext links and form elements that the user can interact with, thereby causing event handlers to be invoked before the second half of the document is loaded.

The fact that event handlers can be invoked before a document is fully loaded has two important implications. First, if your event handler invokes a function, you must be sure that the function is already defined before the handler calls it. One way to guarantee this is to define all your functions in the <HEAD> section of an HTML document. This section of a document is always completely parsed (and any functions in it defined) before the <BODY> section of the document is parsed. Since all objects that define event handlers must themselves be defined in the <BODY> section, functions in the <HEAD> section are guaranteed to be defined before any event handlers are invoked.

The second implication is that you must be sure that your event handler does not attempt to manipulate HTML objects that have not yet been parsed and created. An event handler can always safely manipulate its own object, of course, and also any objects that are defined before it in the HTML file. One strategy is simply to define your web page user interface in such a way that event handlers refer only to objects defined previously. For example, if you define a form that uses event handlers only on the **Submit** and **Reset** buttons, you just need to place these buttons at the bottom of the form (which is where good user-interface style says they should go anyway).

In more complex programs, you may not be able to ensure that event handlers only manipulate objects defined before them, so you need to take extra care with these programs. If an event handler only manipulates objects defined within the same form, it is pretty unlikely that you'll ever have problems. When you manipulate objects in other forms or in other frames, however, this starts to be a real concern. One technique is to test for the existence of the object you want to manipulate before you manipulate it. You can do this simply by comparing it (and any parent objects) to null. For example:

```
<SCRIPT>
function set_name_other_frame(name)
{
  if (parent.frames[1] == null) return;  // Other frame not defined yet
  if (!parent.frames[1].document) return; // Document not loaded in it yet
  if (!parent.frames[1].document.myform) return;    // Form not defined yet
  if (!parent.frames[1].document.myform.name) return; // Field not defined

  parent.frames[1].document.myform.name.value = name;
}
</SCRIPT>

<INPUT TYPE="text" NAME="lastname"
    onChange="set_name_other_frame(this.value)";
>
```

Another technique that an event handler can use to ensure that all required objects are defined involves the onLoad event handler. This event handler is defined in the <BODY> or <FRAMESET> tag of an HTML file and is invoked when the document or frameset is fully loaded. If you set a flag within the onLoad event handler, other event handlers can test this flag to see if they can safely run, with the knowledge that the document is fully loaded and all objects it contains are defined. For example:

```
<BODY onLoad="window.loaded = true;">
 <FORM>
  <INPUT TYPE="button" VALUE="Press Me"
      onClick="if (window.loaded != true) return; doit();"
  >
 </FORM>
</BODY>
```

12.3.3.1. onLoad() and onUnload() event handlers

The onLoad event handler and its partner the onUnload handler are worth a special mention in the context of the execution order of JavaScript programs. Both of these event handlers are defined in the <BODY> or <FRAMESET> tag of an HTML file. (No HTML file can legally contain both these tags.) The onLoad handler is executed when the document or frameset is fully loaded, which means that all images have been downloaded and displayed, all subframes have loaded, any Java applets and plugins (Navigator) have started running, and so on. The onUnload handler is executed just before the page is unloaded, which occurs when the browser is about to move on to a new page. Be aware that when you are working with multiple frames, there is no guarantee of the order in which the onLoad event handler is invoked for the various frames, except that the handler for the parent frame is invoked after the handlers of all its children frames (although this is buggy and doesn't always work correctly in Navigator 2).

The onLoad event handler lets you perform initialization for your web page, while the onUnload event handler lets you undo any lingering effects of the initialization or perform any other necessary cleanup on your page. For example, onLoad could set the Window.defaultStatus property to display a special message in the browser's status bar. Then the onUnload handler would restore the defaultStatus property to its default (the empty string), so that the message does not persist on other pages.

Note that the onUnload event handler should not run any kind of time-consuming operation, nor should it pop up a dialog box. It exists simply to perform a quick cleanup operation; running it should not slow down or impede the transition to a new page.

12.3.4. JavaScript URLs

JavaScript code in a javascript: URL is not executed when the document containing the URL is loaded. It is not interpreted until the browser tries to load the document that the URL refers to. This may be when a user types in a JavaScript URL, or, more likely, when a user follows a link, clicks on a client-side image map, or submits a form. javascript: URLs are usually equivalent to event handlers, and as with event handlers, the code in those URLs can be executed before a document is fully loaded. Thus, you must take the same precautions with javascript: URLs that you take with event handlers to ensure that they do not attempt to reference objects (or functions) that are not yet defined.

12.3.5. JavaScript Entities and Conditional Comments

A JavaScript entity, whether used as the value of an HTML attribute or as part of a conditional comment, must be evaluated as part of the process of HTML parsing. In fact, since the JavaScript code in an entity produces a value that becomes part of the HTML itself, the HTML parsing process is dependent on the JavaScript interpreter in this case. JavaScript entities can always be replaced by more cumbersome scripts that write the affected HTML tags dynamically. Take the following line of HTML:

```
<INPUT TYPE="text" NAME="lastname" VALUE="&{defaults.lastname};">
```

This can be replaced with these lines:

```
<SCRIPT>
  document.write('<INPUT TYPE="text" NAME="lastname" VALUE="' +
            defaults.lastname +
            '">');
</SCRIPT>
```

Similarly, any conditional comment can be replaced by a script that dynamically outputs the HTML content within the comment only if the entity evaluates to true. For all intents and purposes, JavaScript entities and conditional comments are executed just like their equivalent scripts.

12.3.6. Window and Variable Lifetime

A final topic in our investigation of how client-side JavaScript programs run is the issue of variable lifetime. We've seen that the Window object is the global object for client-side JavaScript and all global variables are properties of the Window object. What happens to Window objects and the variables they contain when the web browser moves from one web page to another?

A Window object that represents a top-level browser window exists as long as that window exists. A reference to the Window object remains valid regardless of how many web pages the window loads and unloads. The Window object is valid as long as the top-level window is open.[7]

[7] A Window object may not actually be destroyed when its window is closed. If there are still references to the Window object from other windows, the object is not garbage collected. However, a reference to a window that has been closed is of very little practical use.

A Window object that represents a frame remains valid as long as that frame remains within the frame or window that contains it. If the containing frame or window loads a new document, the frames it originally contained are destroyed in the pro cess of loading that new document.

All this points to the fact that Window objects, whether they represent top-level windows or frames, are fairly persistent. The lifetime of a Window object may be longer than that of the web pages that it contains and displays and longer than the lifetime of the scripts contained in the web pages it displays.

When a web page that contains a script is unloaded because the user has pointed the browser to a new page, the script is unloaded along with the page that contains it. (If the script were not unloaded, a browser might soon be overflowing with various lingering scripts!) But what about the variables defined by the script? Since these variables are actually properties of the Window object that contained the script, you might think that they would remain defined. On the other hand, leaving them defined seems dangerous—a new script that was loaded wouldn't be starting with a clean slate, and in fact, it could never know what sorts of properties (and therefore variables) were already defined.

In fact, all user-defined properties (which includes all variables) are erased whenever a web page is unloaded. The scripts in a freshly loaded document start with no variables defined and no properties in their Window object, except for the standard core and client-side JavaScript properties defined by the system. What this means is that the lifetime of scripts and of the variables they define is the same as the lifetime of the document that contains the scripts. This is potentially much shorter than the lifetime of the window or frame that displays the document containing the scripts. There are important security reasons why this must be so.

The point to remember is that the scripts you write and the variables and functions they define do not and cannot persist across web pages. The programming environment (i.e., the global object) is wiped clean when the browser moves from one web page to another. Every web page begins execution with a clean slate.

URL http://proquest.safaribooksonline.com/1565923928/jscript3-ID-12.3

Additional reading

Safari has identified sections in other books that relate directly to this selection using Self-Organizing Maps (SOM), a type of neural network algorithm. SOM enables us to deliver related sections with higher quality results than traditional query-based approaches allow.

	Section Title	Book Title
1.	Execution of JavaScript Programs	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
2.	Embedding JavaScript in HTML	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
3.	Embedding JavaScript in HTML	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
4.	Event Handlers	JavaScript Design
5.	Client-Side JavaScript: Executable Content in Web Pages	JavaScript: The Definitive Guide, 4th Edition By David Flanagan
6.	Client-Side JavaScript: Executable Content in Web Pages	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
7.	Handling Events	Learn JavaScript ™ In a Weekend ® By Jerry Lee Ford, Jr.
8.	Events and Event Handlers in HTML and JavaScript	JavaScript Design
9.	Event Handlers as JavaScript Properties	JavaScript: The Definitive Guide, 3rd Edition By David Flanagan
10.	JavaScript	HTML & XHTML: The Definitive Guide, 5th Edition

By Bill Kennedy, Chuck Musciano